

MAGAZINE

# BSD

FOR NOVICE AND ADVANCED USERS

## USING FREEBSD AS A FILE SERVER WITH ZFS

BY CARLOS ANTONIO NEIRA BUSTOS

VOL 10 NO 08  
ISSUE 07/2016 (84)  
1898-9144

# CONTENTS

## Module #1

### Introduction 2

Requirements for this workshop.

## Module #2

### Cover the commands and features to administer ZFS volumes 20

We ended the last workshop with a mirror vdev, then we simulated a disk failure in order to understand the “simplicity” of how we could deal with such situations using the ZFS filesystem. Now we will deal with the rest of the daily tasks we would encounter when managing ZFS file systems and commands and ways to do it.

## Module #3

### Armed with the theoretical knowledge acquired in the previous workshops, create a ZFS FreeBSD based File Server. 29

Now we know about RAIDZ, ZIL,cache,vdevs, etc. Now it is time to do something useful with this knowledge and start serving content using the mighty ZFS.

To start exposing our pools to external clients, we will use NFS for our Unix users and Samba for our Windows users.

## About the Instructor

### Carlos Neira 41

## Introduction

---

### Requirements for this workshop:

- FreeBSD production release (10.3).
  - Around 10GB of disk space (we are going to simulate disks using files).
  - At least 4GB of RAM.
  - Root access.
- 

### What is ZFS?

From the handbook itself:

ZFS stands for “Zettabyte file system” it was developed by Sun Microsystems around 2005. It is designed to use a pooled storage method in that space is only used as it is needed for data storage.

It is also designed for maximum data integrity, supporting data snapshots, multiple copies, and data checksums. It uses a software data replication model, known as RAID-Z. RAID-Z provides redundancy similar to hardware RAID, but is designed to prevent data write corruption and to overcome some of the limitations of hardware RAID.

ZFS is a modern 128-bit file system based on the copy-on-write model. It originates from the OpenSolaris project and first appeared in FreeBSD in 2008. ZFS has many innovative features including an integrated volume manager with mirroring and RAID capabilities, data checksumming and compression, writable snapshots that can be transferred between systems and many more. The FreeBSD ZFS filesystem has been updated by merging improvements from illumos project.

When ZFS was shipped in Solaris, it had around 25,000 lines of kernel code and 2,000 of userland code, while Solaris UFS and SVM (Solaris volume manager) together sum up to 90,000 lines of kernel code and 105,000 lines of userland code. ZFS contains more functionality than UFS and SVM with around 1/7 of lines of code.

# Module #1

## Why ZFS?

Here are the main reasons you would choose ZFS as a filesystem for your file server, if you care about your data.

- Highly Scalable

ZFS is a 128 bit filesystem, that means that 256 trillion of zettabytes for storage.

ZFS can handle data pools of  $1.84 \times 10^{19}$  times more data than 64-bit systems.

That is a lot of data that you could gather around a lifetime. Imagine one directory could have up to

281474976710656 files whose size could be up to 16 exabytes each.

- Data integrity

ZFS is a transactional file system, that means that the data is always consistent on disk. For example, in other filesystems, if your machine shutdowns unexpectedly just in the moment a block of data is written, the filesystem is now in an inconsistent state and the solution for this problem was to fsck the disk. This does not happen to ZFS due to its copy on write model.

- Snapshots and replication

ZFS allows for snapshots to be taken on a singular or periodic basis and allows you to backup individual datasets as often as needed. Snapshots allow for a simple rollback to prior states in case of file deletion or system instability.

- Easy administration

For system administrators, the greatest improvement of ZFS over traditional filesystems is the ease of administration. All operations on ZFS are done through commands `zfs` and `zpool`.

## Installing FreeBSD

Download one of the following images, depending on your architecture:

<http://ftp.freebsd.org/pub/FreeBSD/releases/ISO-IMAGES/10.3/FreeBSD-10.3-RELEASE-amd64-dvd1.iso>

<http://ftp.freebsd.org/pub/FreeBSD/releases/ISO-IMAGES/10.3/FreeBSD-10.3-RELEASE-i386-dvd1.iso>

# Module #1

## USB images:

<http://ftp.freebsd.org/pub/FreeBSD/releases/ISO-IMAGES/10.3/FreeBSD-10.3-RELEASE-i386-mini-memstick.img>

To write the USB image to a stick, on Unix you could use `dd`, man `dd` on

```
# dd if=FreeBSD-10.3-RELEASE-amd64-memstick.img of=/dev/da0 bs=1M
conv=sync
```

on Windows you will need <https://launchpad.net/win32-image-writer/>

If you are going to use an i386 system for this workshop, you need to follow the steps from the ZFS Tuning Guide (<https://wiki.freebsd.org/ZFSTuningGuide>) to avoid kernel panics.

## i386

Typically, you need to increase `vm.kmem_size_max` and `vm.kmem_size` (with `vm.kmem_size_max >= vm.kmem_size`) to not get kernel panics (kmem too small). The value depends upon the workload. If you need to extend them beyond 512M, you need to recompile your kernel with increased `KVA_PAGES` option, e.g. add the following line to your kernel configuration file to increase available space for `vm.kmem_size` beyond 1 GB:

```
options KVA_PAGES=512
```

To choose a good value for `KVA_PAGES` read the explanation in the `sys/i386/conf/NOTES` file.

By default, the kernel receives 1 GB of the 4 GB of address space available on the i386 architecture, and this is used for all of the kernel address space needs, not just the kmem map. By increasing `KVA_PAGES`, you can allocate a larger proportion of the 4 GB address space to the kernel (2 GB in the above example), allowing more room to increase `vm.kmem_size`. The trade-off is that user applications have less address space available, and some programs (e.g. those that rely on mapping data at a fixed address that is now in the kernel address space, or which require close to the full 3 GB of address space themselves) may no longer run. If you change `KVA_PAGES` and the system reboots (no panic) after running a while, this may be because the address space for userland applications is too small now.

# Module #1

For *really* memory constrained systems, it is also recommended to strip out as many unused drivers and options from the kernel (which will free a couple of MB of memory). A stable configuration with `vm.kmem_size="1536M"` has been reportedly using an unmodified 7.0-RELEASE kernel, relatively sparse drivers as required for the hardware and options `KVA_PAGES=512`.

Some workloads need greatly reduced ARC size and the size of VDEV cache. ZFS manages the ARC through a multi-threaded process. If it requires more memory for ARC, ZFS will allocate it. Previously, it exceeded `arc_max` (`vfs.zfs.arc_max`) from time to time, but with 7.3 and 8-stable as of mid-January 2010 this is not the case anymore. On memory constrained systems, it is safer to use an arbitrarily low `arc_max`. For example, it is possible to set `vm.kmem_size` and `vm.kmem_size_max` to 512M, `vfs.zfs.arc_max` to 160M, keeping `vfs.zfs.vdev.cache.size` to half its default size of 10 Megs (setting it to 5 Megs can even achieve better stability, but this depends upon your workload).

There is one example (CySchubert) of ZFS running nicely on a laptop with 768 Megs of physical RAM with the following settings in `/boot/loader.conf`:

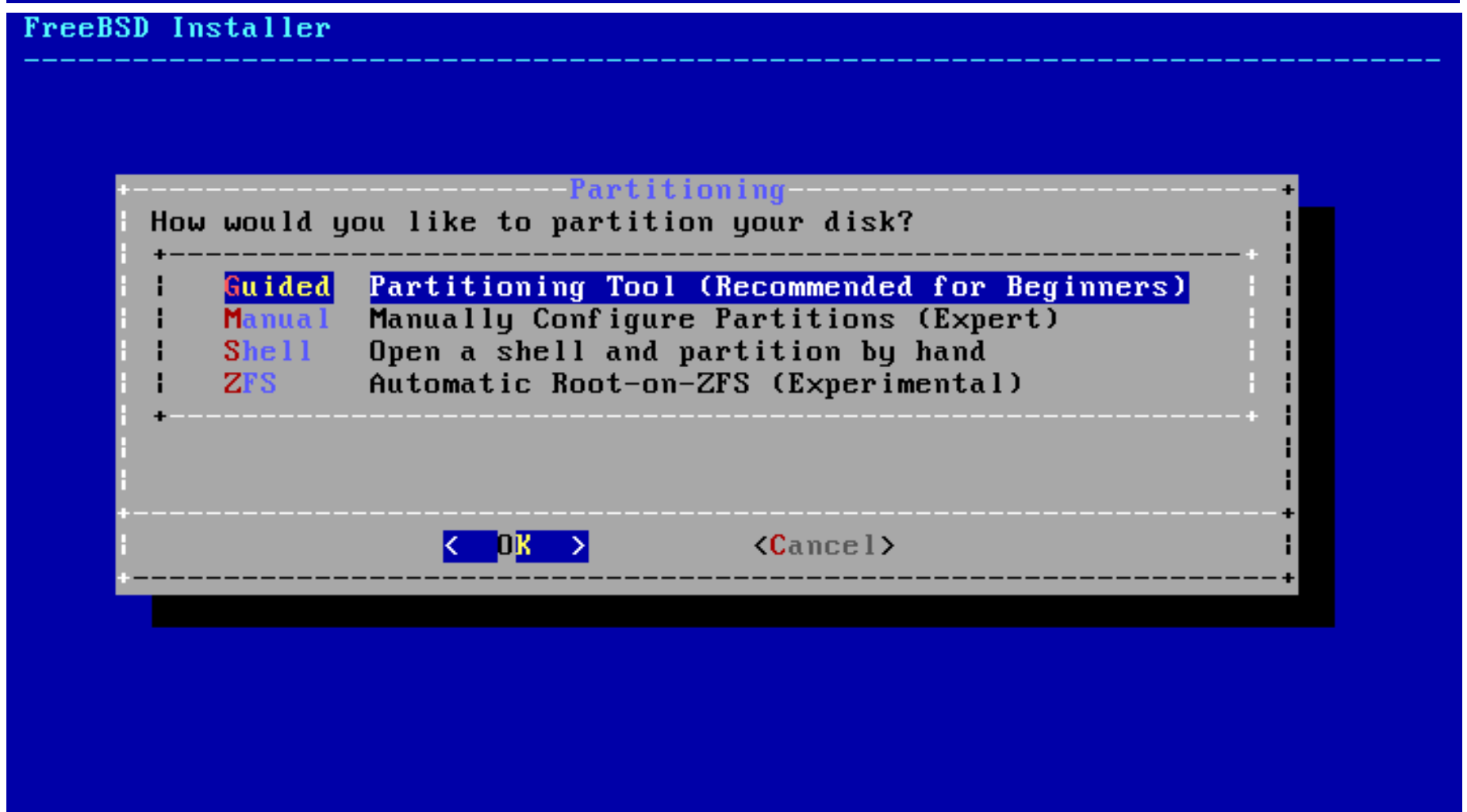
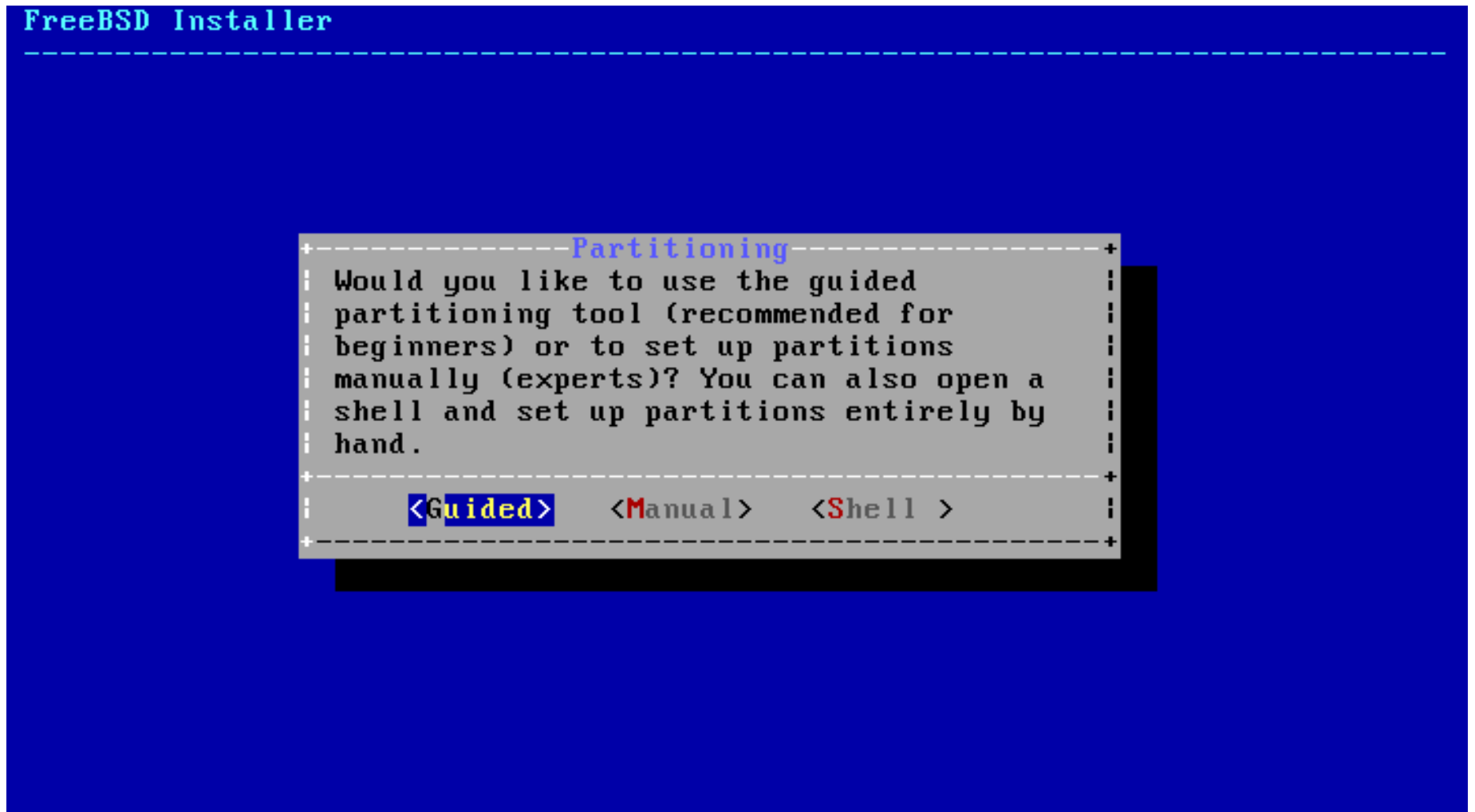
- `vm.kmem_size="330M"`
- `vm.kmem_size_max="330M"`
- `vfs.zfs.arc_max="40M"`
- `vfs.zfs.vdev.cache.size="5M"`

If you have not installed FreeBSD before, don't worry, it's really easy, just follow this guide from the handbook <https://www.freebsd.org/doc/handbook/bsdinstall.html>

As our focus is ZFS on FreeBSD, when you are on this screen

<https://www.freebsd.org/doc/handbook/bsdinstall-partitioning.html>

# Module #1



Choose guided and then ZFS Automatic-Root-on-ZFS (don't worry, it's not experimental).

# Module #1

## Basic concepts

### Zpool

A logical group of devices describing the layout and physical characteristics of the available storage. Disk space for datasets is allocated from a pool.

A ZFS pool is a storage object consisting of virtual devices. These `vdevs` can be:

- Disk (partition, GEOM object, ...)
- File (experimental purposes)
- Mirror (groups of two or more vdevs)
- RAIDz, RAIDz2, RAIDz3 (single to triple parity RAIDz)
- Spare (pseudo-vdev for hot spares)
- Log (separate ZIL device, may not be RAIDz)
- Cache (L2 cache, may not be mirror or RAIDz)

We will take a look at each vdev later in these workshops.

ZFS support is self healing but only if your vdevs provide redundancy that is done on how you organize your disks. For example, a vdev composed of a single drive is called stripe (in fact you could compose the stripe of many single disk vdevs) and ZFS writes along the stripe; the vdev is in charge of providing redundancy. In the case of a stripe, if the disk fails, your entire pool goes down.

You could check the pools available in your system using `ZPOOL(8)`. In a fresh FreeBSD installation, you should see something like this if you used automatic root on ZFS at the install.

```
$ zpool list
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
zroot    7.94G  1.95G   5.99G        -    19%   24%   1.00x  ONLINE  -
$
```

Let's explain what each column represents:

**NAME** - The name of the pool.

**SIZE** - The total size of the pool, equal to the sum of the sizes of all top-level virtual devices.

**ALLOC** - The amount of physical space allocated to all datasets and internal metadata. Note

# Module #1

this amount differs from the amount of disk space as reported at the file system level.

**FREE** -The amount of unallocated space in the pool.

**CAP (CAPACITY)** - The amount of disk space used, as a percentage of the total disk space.

**HEALTH** - The current health status of the pool.

**ALTROOT** - The alternate root of the pool, if one exists.

## Dataset

Is a generic name for the following ZFS components: clones, filesystems, snapshots, and volumes.

Each dataset is identified by a unique name in the ZFS namespace. Datasets are identified using the following format:

```
pool/path[@snapshot]
```

### pool

Identifies the name of the storage pool that contains the dataset.

### path

Is a slash-delimited path name for the dataset component.

### snapshot

Is an optional component that identifies a snapshot of a dataset.

You could think of a dataset as a plain partition, if that puts your mind at ease.

## Snapshot

A read-only copy of a filesystem or volume at a given point in time. Snapshots are created using the command `zfs snapshot <name of snapshot>`. For example, to take a snapshot of `usr` dataset type:

```
# zfs snapshot zroot/user@now
```

# Module #1

## Clone

A filesystem whose initial contents are identical to the contents of a snapshot. To create a clone, first we need a snapshot. What is the difference between clone and snapshot? Snapshots are read only, clones are read/write.

Clones can only be created from a snapshot. When a snapshot is cloned, an implicit dependency is created between the clone and snapshot. Even though the clone is created somewhere else in the dataset hierarchy, the original snapshot cannot be destroyed as long as the clone exists. The origin property exposes this dependency, and the `zfs destroy` command lists any such dependencies, if they exist. To create a clone we use the `zfs clone` command.

```
# zfs snapshot mypool/sfw/production@yesterday  
  
# zfs clone mypool/sfw/production@yesterday mypool/devs/  
production/release
```

## Mirror

A virtual device that stores identical copies of data on two or more disks. If any disk in a mirror fails, any other disk in that mirror can provide the same data. Very good for random and sequential reads as data is fetched from both disks, but write as in read operation must be performed in both disks. If one disk is slower than the other, you have a bottleneck there.

## RAID levels

This is good to know to refresh our knowledge before defining what a RAIDZ is.

- **RAID-0 Data** is striped across devices for maximal write performance. It is an outlier among the other RAID levels as it provides no actual data protection.
- **RAID-1 Disks** are organized into mirrored pairs and data is duplicated on both halves of the mirror. This is typically the highest-performing RAID level, but at the expense of lower usable capacity.
- **RAID-2** Data is protected by memory-style ECC (error correcting codes). The number of parity disks required is proportional to the log of the number of data disks.
- **RAID-3** Protection is provided against the failure of any disk in a group of  $N+1$  by carving up blocks and spreading them across the disks — bitwise parity. Parity resides on a single disk.
- **RAID-4** A group of  $N+1$  disks is maintained such that the loss of any one disk would not result in data loss. A single disk is designated as the dedicated parity disk. Not all disks participate in

# Module #1

reads (the dedicated parity disk is not read except in the case of a failure). Typically, parity is computed simply as the bitwise XOR of the other blocks in the row.

- **RAID-5 N+1** redundancy as with RAID-4, but with distributed parity so that all disks participate equally in reads.
- **RAID-6** This is like RAID-5, but employs two parity blocks, P and Q, for each logical row of N+2 disk blocks.
- **RAID-7** Generalized M+N RAID with M data disks protected by N parity disks (without specifications regarding layout, parity distribution, etc.).
- **RAID-0** Data is striped across devices for maximal write performance. It is an outlier among the other RAID levels as it provides no actual data protection.

## Write Hole

The “write hole” effect can happen if a power failure occurs during the write. It happens in all the array types, including, but not limited to, RAID5, RAID6, and RAID1. In this case, it is impossible to determine which data blocks or parity blocks have been written to the disks and which have not. In this situation, the parity data does not match the rest of the data in the stripe. Also, you cannot determine with confidence which data is incorrect – parity or one of the data blocks.

## RAIDZ

RAID-Z is a data/parity scheme like RAID-5, but it uses dynamic stripe width. Every block is its own RAID-Z stripe, regardless of blocksize. This means that every RAID-Z write is a full-stripe write. This, when combined with the copy-on-write transactional semantics of ZFS, completely eliminates the RAID write hole. RAID-Z is also faster than traditional RAID because it never has to do read-modify-write. RAIDZ spreads parity information across all the disks. If a disk fails, RAIDZ has the parity information to recalculate the missing data.

## RAIDZ1

Resists the failure of single disk device (has one parity disk), if a second disk fails before the first failing one is replaced, all data is gone.

## RAIDZ2

Like RAIDZ1 but has two parity disks per vdev, it endures the failure of two disks so you have plenty of time to replace the faulty ones. Performance is worse than RAIDZ1.

## RAIDZ3

Like RAIDZ2. It can endure the failure of three disks of your five disk array. Needs three disks for parity.

# Module #1

## Resilvering and scrub

ZFS has no fsck repair tool equivalent, common on Unix filesystems, which does filesystem validation and file system repair. Instead, ZFS has a repair tool called “scrub” which examines and repairs silent corruption and other problems. Some differences are:

- fsck must be run on an offline file system, which means the filesystem must be unmounted and is not usable while being repaired.
- scrub does not need the ZFS filesystem to be taken offline; scrub is designed to be used on a mounted, live filesystem.
- fsck usually only checks metadata (such as the journal log) but never checks the data itself. This means, after an fsck, the data might still be corrupt.
- scrub checks everything, including metadata and the data. The effect can be observed by comparing fsck to scrub times – sometimes an fsck on a large RAID completes in a few minutes, which means only the metadata was checked. Traversing all metadata and data on a large RAID takes many hours, which is exactly what scrub does.

The official recommendation from Sun/Oracle is to scrub enterprise-level disks once a month, and cheaper commodity disks once a week.

Scrubbing takes time; you should create a job that scrubs disks at night or when the system is not being used.

The purpose of this tutorial is to explore some ZFS functionalities in a safe way. To grasp the power and flexibility of this filesystem, we will take a look at the basic functionalities:

- Create a ZFS pool
- Create a ZFS mirror
- Simulate a failure on a mirrored disk
- Replace a disk
- Adding disks to a mirrored zpool
- Check I/O on ZFS pools

## ZFS properties

Properties are divided into two types, native properties and user-defined (or “user”) properties.

# Module #1

Native properties either export internal statistics or control ZFS behavior. In addition, native properties are either editable or read-only. User properties have no effect on ZFS behavior, but you can use them to annotate datasets in a way that is meaningful in your environment.

User property names must conform to the following conventions:

- They must contain a colon (':') character to distinguish them from native properties.
- They must contain lowercase letters, numbers, or the following punctuation characters: '.', '+', '-', '\_'.
- The maximum length of a user property name is 256 characters.

For example, let's add a property to a dataset. First create the dataset, then set a property to it, as in the following screenshot.

```
$ sudo zfs create exampleMirror/dev01
Password:
$ zfs set team:users=devs exampleMirror/dev01
cannot set property for 'exampleMirror/dev01': permission denied
$ sudo zfs set team:users=devs exampleMirror/dev01
$ zfs list
NAME                                USED  AVAIL  REFER  MOUNTPOINT
exampleMirror                       240K  79.8M   148K   /exampleMirror
exampleMirror/dev01                 19K  79.8M    19K   /exampleMirror/dev01
workshop1                         1.07M  78.9M   1.02M   /workshop1
zroot                             2.76G  4.93G    96K   /zroot
zroot/R00T                        1.60G  4.93G    96K   none
zroot/R00T/default                1.60G  4.93G   1.60G   /
zroot/tmp                          96K  4.93G    96K   /tmp
zroot/usr                        1.16G  4.93G    96K   /usr
zroot/usr/home                    548K  4.93G   548K   /usr/home
zroot/usr/ports                   642M  4.93G   642M   /usr/ports
zroot/usr/src                     547M  4.93G   547M   /usr/src
zroot/var                         660K  4.93G    96K   /var
zroot/var/audit                   96K  4.93G    96K   /var/audit
zroot/var/crash                   96K  4.93G    96K   /var/crash
zroot/var/log                     176K  4.93G   176K   /var/log
zroot/var/mail                    100K  4.93G   100K   /var/mail
zroot/var/tmp                      96K  4.93G    96K   /var/tmp
$ sudo zfs get -r team:devs exampleMirror
NAME                                PROPERTY  VALUE  SOURCE
exampleMirror                      team:devs -      -
exampleMirror/dev01                team:devs -      -
$
```

Adding metadata allows you to create schemes for scrubbing, backing up, etc., based on properties.

# Module #1

## Creating Disks and Pools

To try some ZFS features, first we need to create pools. We will use files to simulate real disks so we can test things safely. I will use `mkfile` utility to create some files and use those as disks.

`mkfile` creates one or more files that are suitable for use as NFS-mounted swap areas, or as local swap areas. The file is padded with zeros by default. The default size is in bytes, but it can be flagged as exabytes, petabytes, terabytes, gigabytes, megabytes, kilobytes, or blocks, with the `e`, `p`, `t`, `g`, `m`, `k`, or `b` suffixes, respectively.

Now let's create some disks!

```
# mkdir array
# cd array
# mkfile 128m disk00 disk01 disk02 disk03
# ls -lrt

total 524416

-rw----- 1 root wheel 134217728 Jul 29 21:51 disk00
-rw----- 1 root wheel 134217728 Jul 29 21:51 disk01
-rw----- 1 root wheel 134217728 Jul 29 21:51 disk02
-rw----- 1 root wheel 134217728 Jul 29 21:51 disk03
```

Here I'm creating four disks of 128mb each, as you can see in the `ls` output.

If you don't have the `mkfile` utility, install it by typing.

```
# pkg install mkfile
```

## Creating Zpools

All ZFS filesystems live in a pool, so first we need to create a zpool. We can check pools with the `zpool` command. Before creating new zpools, you should check for existing zpools to avoid confusing them with your tutorial pools. You can check what zpools exist with `zpool list`:

```
# zpool list
```

# Module #1

Now let's create a zpool with `zpool create`.

```
# zpool create tutorial /array/disk00
```

List the current pools:

```
# zpool list

NAME SIZE ALLOC FREE CAP DEDUP HEALTH ALTROOT
tutorial 123M 77K 123M 0% 1.00x ONLINE -
```

Now let's use the pool. Create a new file on the pool we just created.

```
# mkfile 1m /tutorial/file1

# ls -lh /tutorial

total 1027

-rw----- 1 root wheel 1.0M Jul 29 22:24 file1
```

Here I have created a 1MB file on the newly created zpool.

## Creating a mirror vdev to provide redundancy

A pool with only one disk doesn't offer any redundancy. Let's create a new zpool called `example2` using a couple of disks. We will use the keyword `mirror`. As the name states, it will make a mirror using this pair of disks when we create the zpool.

```
# zpool create example2 mirror /array/disk00 /array/disk01

# zpool list

NAME SIZE ALLOC FREE CAP DEDUP HEALTH ALTROOT
example2 123M 91K 123M 0% 1.00x ONLINE -
```

# Module #1

We can check the status of our pools with the `zpool status` command.

```
# zpool status

pool: example2

state: ONLINE

scan: none requested

config:

NAME STATE READ WRITE CKSUM
example2 ONLINE 0 0 0
mirror-0 ONLINE 0 0 0
/array/disk00 ONLINE 0 0 0
/array/disk01 ONLINE 0 0 0

errors: No known data errors
```

Let's create a file again and check the status after that (I'll create a 32MB file).

```
# mkfile 32m /example2/file

NAME SIZE ALLOC FREE CAP DEDUP HEALTH ALTROOT
example2 123M 32.8M 90.2M 26% 1.00x ONLINE -
```

So now we have our data stored over the two disks, remember that redundancy is at the vdev level and so creating a vdev that offers redundancy ZFS could self heal.

Let's burn down the data center!

Now everything is nice and calm, but sometimes bad things happen to good people, like a disk going bad at 3 am, or your ebook collection being lost due to a disk failure.

Let's simulate a disk failure; for that I'll overwrite the first disk label with random data.

```
# dd if=/dev/random of=/array/disk01 bs=1024 count=1
```

# Module #1

```
1+0 records in
1+0 records out
1024 bytes transferred in 0.029959 secs (34180 bytes/sec)
```

In case you don't know about the dd command, here is what it does:

***“The dd utility copies the standard input to the standard output. Input data is read and written in 512-byte blocks. If input reads are short, input from multiple reads are aggregated to form the output block. When finished, dd displays the number of complete and partial input and output blocks and truncated input records to the standard error output.”***

So what I did was to write one time a block size of 1024 bytes from /dev/random to our disk01.

ZFS automatically checks for errors when it reads/writes files. We can force a check with the scrub command.

```
# zpool scrub example2

# zpool status

pool: example2

state: DEGRADED

status: One or more devices could not be used because the label is
missing or

invalid. Sufficient replicas exist for the pool to continue
functioning in a degraded state.

action: Replace the device using 'zpool replace'.

see: http://illumos.org/msg/ZFS-8000-4J

scan: scrub repaired 0 in 0h0m with 0 errors on Mon Jul 29 23:06:53
2013

config:

NAME STATE READ WRITE CKSUM
```

# Module #1

```
example2 DEGRADED 0 0 0

mirror-0 DEGRADED 0 0 0

/array/disk00 ONLINE 0 0 0

2742290791000160517 UN

errors: No known data errors
```

We messed up the disk so it shows as UNAVAIL, but no errors are reported for the pool as a whole:

***“Sufficient replicas exist for the pool to continue functioning in a degraded state.”***

We still can read and write to the pool.

```
root@diego:/ # ls -lrt /example2/

total 32779

-rw----- 1 root wheel 33554432 Jul 29 22:41 file
```

## Replacing the faulty disk

Let's take out the bad disk from the pool using the detach command.

```
# zpool detach example2 /array/disk01
```

Now let's erase our file and create a new one to simulate a new disk.

```
# rm /array/disk01

# mkfile 128m /array/disk01
```

To attach another device, we specify an existing device in the mirror to attach it to with zpool attach:

```
# zpool attach example2 /array/disk00 /array/disk01

# zpool status example2
```

# Module #1

```
pool: example2

state: ONLINE

scan: resilvered 54.4M in 0h0m with 0 errors on Tue Jul 30 09:15:26
2013

config:

NAME STATE READ WRITE CKSUM
example2 ONLINE 0 0 0
mirror-0 ONLINE 0 0 0
/array/disk00 ONLINE 0 0 0
/array/disk01 ONLINE 0 0 0

errors: No known data errors
```

If you type `zpool status` fast enough, after you attach the new disk you will see a resilver (remirror-ing) in progress with `zpool status`. Once the resilver is complete, the pool is healthy again (you can also use `ls` to check the files are still there):

```
# ls /example2/

file file2
```

---

## References

1. (n.d.). Retrieved July 02, 2016, from [https://en.wikipedia.org/wiki/Copy-on-write#Copy-on-write\\_in\\_storage\\_media](https://en.wikipedia.org/wiki/Copy-on-write#Copy-on-write_in_storage_media)
2. “Write hole” phenomenon. (n.d.). Retrieved July 03, 2016, from <http://www.raid-recovery-guide.com/raid5-write-hole.aspx>

# Module #1

3. 19.2. Quick Start Guide. (n.d.). Retrieved July 03, 2016, from <https://www.freebsd.org/doc/handbook/zfs-quickstart.html>
4. Chapter 7 Working With Oracle Solaris ZFS Snapshots and Clones. (n.d.). Retrieved July 02, 2016, from <http://docs.oracle.com/cd/E19253-01/819-5461/gavvx/index.html>
5. FreeBSD Man Pages. (n.d.). Retrieved July 03, 2016, from [https://www.freebsd.org/cgi/man.cgi?zfs\(8\)](https://www.freebsd.org/cgi/man.cgi?zfs(8))
6. Introducing ZFS Properties. (n.d.). Retrieved July 03, 2016, from <http://docs.oracle.com/cd/E19253-01/819-5461/6n7ht6r2s/index.html#gcfgr>
7. Nested RAID levels. (n.d.). Retrieved July 03, 2016, from [https://en.wikipedia.org/wiki/Nested\\_RAID\\_levels](https://en.wikipedia.org/wiki/Nested_RAID_levels)
8. Querying ZFS Storage Pool Status. (n.d.). Retrieved July 02, 2016, from <http://docs.oracle.com/cd/E19253-01/819-5461/gaynp/index.html>
9. ZFS Best Practices Guide. (n.d.). Retrieved July 02, 2016, from [http://www.solarisinternals.com/wiki/index.php/ZFS\\_Best\\_Practices\\_Guide](http://www.solarisinternals.com/wiki/index.php/ZFS_Best_Practices_Guide)

## Cover the commands and features to administer ZFS volumes

We ended the last workshop with a mirror vdev, then we simulated a disk failure in order to understand the “simplicity” of how we could deal with such situations using the ZFS filesystem. Now we will deal with the rest of the daily tasks we would encounter when managing ZFS file systems and commands and ways to do it.

### Creating pools and vdevs

When we define a pool as a storage object composed of virtual devices, it means when we create a pool we need to specify which vdevs are going to be part of it.

- Disk
- File
- Mirror
- RAIDZ
- Spare
- Log
- Cache

### Single disk pool

Let's start creating a pool from a disk; you can add more disks if you need more space. Remember, as this vdev does not provide redundancy, ZFS cannot self heal.

```
# zpool create mypool disk1
```

# Module #2

## Multiple disks pool

When we create a pool just specifying the disks is called a striped pool; this pool has no redundancy but could have lots of disk space.

```
#zpool create mypool <disk0 ....diskn -1>
```

## Single disk pool using a file

A single pool can be created with a file, the same way we did before to experiment with ZFS:

```
# zpool create myfilepool <path to file>
```

## Mirror

A mirrored pool as we created in the last workshop, this vdev offers redundancy and ZFS could self heal this pool.

```
# zpool create Mpool mirror disk0 disk1
```

## RAIDZ pools

To create a RAIDZ vdev, we just use the `raidz<1-3>` keyword to indicate to `zpool` that we want to build a vdev of that nature. The redundancy and performance of a mirror vdev is good but we lose too much disk space of our disks; the complexity offered by a RAIDZ and the disk usage make it a very good choice comparing it to a mirror, if you need more disk space.

```
# zpool create rzpool raidz1 disk0 disk1 disk_for_parity
```

## RAIDZ1

Remember, RAIDZ uses one disk for parity. That does not mean that this disk has all the parity information, the parity is stored across the devices. A conventional RAID system uses full width stripes, where all stripes span all disks. When you add another disk, the RAID system has to change how all of the existing data is laid out to preserve this full width; it goes from having the data and parity striped across  $N$  disks to having it striped across  $N+1$  disks. But with variable width stripes, ZFS doesn't have this problem; adding an existing disk doesn't require touching any of the existing stripes, even what were full width stripes. All that happens is they go from being full width stripes to being partial width stripes.

# Module #2

## RAIDZ2

As we stated in the previous workshop, RAIDZ2 needs two disks for parity.

```
# zpool create rzpool2 raidz2 disk00 disk01 pdisk0 pdisk1
```

## RAIDZ3

As we stated in the previous workshop RaidZ3 needs 3 disks for parity.

```
# zpool create rzpool3 raidz2 disk00 disk01 disk02 pdisk00 pdisk01  
pdisk02
```

The rule for creating RAIDZ arrays is the following:

```
2 x ( disk quantity ) + parity_required_disks = amount_of_disks_need-  
ed_for_your raid-z array.
```

We have only created pools using a single vdev, but a pool could include multiple vdevs; adding vdevs increases the space available and the performance as a pool splits all writes between vdevs. To create a pool with multiple vdevs we just use the zpool command specifying the first vdev to be created and the required disks and then the next vdev type and its disks, for example:

```
# zpool create multipool mirror disk00 disk01 mirror disk02 disk03
```

Now this pool has two vdevs and remember that writes are split through the vdevs so this configuration is a RAID 10. Let's do a quick refresh on multi level RAIDs.

A Multi-level RAID combines two (or more) levels of RAID algorithms into a single addressable logical unit. So, in the previous example, we have combined RAID1(mirror) with a RAID0 (stripe) which results in RAID10 configuration. Let's take a look at the following multilevel raid configurations:

- RAID 0+1 (Mirror of Stripes, RAID 01, or RAID 0 then RAID 1)

RAID 0+1 is a mirror (RAID 1) of a stripe set (RAID 0).

# Module #2

We have only created pools using a single vdev, but a pool could include multiple vdevs; adding

- RAID 10 (Stripe of Mirrors, RAID 1+0, or RAID 1 then RAID 0)

RAID 10 is a stripe (RAID 0) of multiple mirror sets (RAID 1).

- RAID 50 (Stripe of Parity Set, RAID 5+0, or RAID 5 then RAID 0)

RAID 50 is a stripe (RAID 0) of multiple parity sets (RAID 5).

Let's create a pool of each of these multilevel schemes using RAIDZ vdevs.

## RAID 0+1

As ZFS always stripes across vdevs, this one is not achievable.

## RAID 1+0

As the naming convention of multilevel RAIDs establishes, let's first create a mirror vdev to be striped across another mirror vdev.

```
# zpool create raidz10 mirror disk00 disk01 mirror disk02 disk03
```

## RAID 5+0

For this, we just create a RAID5 that is equivalent to a multiple RAIDZ1 vdev configuration.

```
# zpool create raid50 raidz1 disk00 disk01 paritydisk0 raidz1  
disk02 disk03 paritydisk1
```

## Hot Spare vdev

ZFS allows devices to be associated with pools as “hot spares”. These devices are not actively used in the pool, but when an active device fails, it is automatically replaced by a hot spare. To create a pool with hot spares, specify a “spare” vdev with any number of devices. Unfortunately, in FreeBSD 10, the ZFS spare functionality basically just marks the drive as a spare, so you, as the admin, know what drive to use when replacement time comes. This feature is available only in current version at the moment.

To mark a disk as a spare in a RAIDZ vdev, the keyword spare is used.

```
# zpool create pool raidz da1 da2 da3 spare da4
```

# Module #2

## Log vdev (ZIL ZFS Intent log)

All file system related system calls are logged as transaction records by the ZIL. These transaction records contain sufficient information to replay them back in the event of a system crash.

The ZIL acts like a write cache; adding a dedicated write cache to your pool could improve performance, and you will want fast devices to act as cache.

To add a log vdev to a pool, we type the following incantation:

```
# zpool add mypool log diskforlog
```

When creating a pool you can attach a write cache to it.

```
# zpool add mypool raidz1 disk00 disk01 diskparity log disklog
```

Beware, if the ZIL fails it could cause data loss as transactions could not be reproduced in case of a system crash, so we will want to provide redundancy to this vdev like mirroring. If the ZIL is not mirrored, and the ZIL drive fails, the system will revert now to write data right into the disk, severely hampering performance, we don't want this so we do the following to be safe:

```
# zpool create raidz1 disk00 disk01 diskparity log mirror diskzil0  
diskzil01
```

## Cache vdev

This vdev, as the name states, is a cache but in this case, instead of a write cache, it performs as a read cache improving your performance; it is not so dangerous if your disk performing as a disk cache fails, ZFS will start reading from the actual pool in that case, so it makes little sense adding redundancy.

Like zil the command is the following:

```
#zpool create raidz1 disk00 disk01 diskparity cache diskcache00
```

# Module #2

## Understanding ZFS properties

In ZFS, there exists two types of properties; there are “native” properties, these could affect performance in your pool, and the “user defined” properties that are meant to be used and created by system administrators.

Some native properties are dataset-specific. Native properties have access to information on the internal operations and state of the ZFS system. User defined properties allow annotation, customization, and automation of the storage system based on the metadata added.

ZFS properties are inherited. Every time you create a new dataset under a pool or another dataset, the new dataset will have the same properties as its parent. You can override those properties as needed at creation. This might be important, for example, when you have a user requirement for a file system-specific configuration, such as enabling a high-level of compression for archival purposes.

Properties that provide statistical information are read-only. Properties that affect the operation of a storage system are read/write in order to change the behavior.

Let’s examine the properties of the following dataset.

## Querying ZFS properties

To obtain information about properties, native or user defined ones, the parameter `get` is used. For example, if we want to check if a pool has dedup feature turned on, we need to type the following command:

```
# zfs get dedup mypool
```

If we want to set a value for a read/write property we just set it:

```
# zfs set dedup=on mypool
```

if you need to see all the properties associated with a dataset:

```
# zfs get all mypool
```

Keep in mind that there are three properties that cannot be changed once set. These properties are related to file names and include case sensitivity (`casesensitivity`), unicode normalization (`normalization`), and UTF-8 required (`utf8only`). These properties have to be set when the file system is created using the `zfs create` or `zpool create` commands and cannot be changed later.

# Module #2

To set the properties at creation time, we type:

```
# zfs create -o casesensitivity=insensitive -o normalization=formC  
mypool/mydataset
```

Multiple properties can be passed during the creation of a file system. The created mypool/myfs file system will have file name case insensitivity and use a specific method for file name normalization. These properties can no longer be changed for mypool/mydataset.

## Native properties

There are a couple of native properties that affect your pool performance and space available; these are handy as we are planning on building a file server. Those are dedup and compression.

### Dedup

```
# zfs set dedup=on mypool
```

if you turn on dedup (deduplication) in a dataset, ZFS will check for duplicate data and store it only once. To do that, a dedup table is created in memory and it is consulted every time a file is accessed. If your rate of duplication is high and your memory low, then the table would be dumped to disk and ZFS will start reading that table from disk, slowing all to a crawl. Use this option only if you have plenty of memory and have analyzed your data and this will be a benefit in the long term.

### Compression

```
# zfs set compression=on mypool
```

The other property is compression. As the name gives away, ZFS will compress files in real time, there is no need to compress log files to save space or such scripts. But this is not free; it will cause some CPU overhead, depending on the algorithm we choose to compress our datasets. The algorithms available are: LZJB (default), LZ4 and gzip (levels 1-9, 1 being the one that has best performance but not so great compression to 9 the slowest one, if you omit the value it defaults to 6). LZ4 is the modern one and the one that presents the best performance. Choose LZ4 or, if you have a special use case, do some testing with your data and a compression enabled pool. Remember that when you set the compression property, it will only affect the new files created on the dataset.

# Module #2

Let's set gzip level 1 compression on a dataset and then change it to lz4:

```
# zfs set compression=gzip-1 zroot/var/log
# zfs set compression=lz4 zroot/var/log
```

## Dataset Quota

Dataset quotas are used to restrict the amount of space that can be consumed by a particular dataset. Reference Quotas work in very much the same way, but only count the space used by the dataset itself, excluding snapshots and child datasets. Similarly, user and group quotas can be used to prevent users or groups from using all of the space in the pool or dataset.

### Set a user quota:

```
# zfs set quota=1G pool1/export/home/user1
```

### Set a reference quota:

```
# zfs set refquota=500M pool1/export/home/user1
```

---

## References

19.4. zfs Administration. (n.d.). Retrieved July 05, 2016, from <https://www.freebsd.org/doc/handbook/zfs-zfs.html>

19.4. zfs Administration. (n.d.). Retrieved July 05, 2016, from <https://www.freebsd.org/doc/handbook/zfs-zfs.html>

Displaying ZFS Delegated Permissions (Examples) – Oracle Solaris ZFS Administration Guide. (n.d.). Retrieved July 05, 2016, from [https://docs.oracle.com/cd/E23823\\_01/html/819-5461/gebww.html](https://docs.oracle.com/cd/E23823_01/html/819-5461/gebww.html)

Exploring ZFS Properties – ikawnoclastic thoughts. (2015, April 30). Retrieved July 05, 2016, from <https://ikawnoclast.com/systems/zfs/exploring-zfs-properties/>

Get the basics on multilevel RAID sets – TechRepublic. (n.d.). Retrieved July 04, 2016, from <http://www.techrepublic.com/article/get-the-basics-on-multilevel-raid-sets/>

# Module #2

Introducing ZFS Properties. (n.d.). Retrieved July 05, 2016, from <http://docs.oracle.com/cd/E19253-01/819-5461/gazss/index.html>

L2ARC Compression. (n.d.). Retrieved July 05, 2016, from <http://wiki.illumos.org/display/illumos/L2ARC> Compression

LZ4 Compression. (n.d.). Retrieved July 05, 2016, from <http://wiki.illumos.org/display/illumos/LZ4> Compression

Optimizing MySQL Performance with ZFS – Slides available (Neelakanth Nadgir's blog). (n.d.). Retrieved July 04, 2016, from [https://blogs.oracle.com/realneel/entry/optimizing\\_mysql\\_performance\\_with\\_zfs](https://blogs.oracle.com/realneel/entry/optimizing_mysql_performance_with_zfs)

Revision 300906. (n.d.). Retrieved July 04, 2016, from <https://svnweb.freebsd.org/base?view=revision&revision=300906>

The ZFS Intent Log (Neelakanth Nadgir's blog). (n.d.). Retrieved July 04, 2016, from [https://blogs.oracle.com/realneel/entry/the\\_zfs\\_intent\\_log](https://blogs.oracle.com/realneel/entry/the_zfs_intent_log)

ZFS: The Lumberjack (Neil Perrin's Weblog). (n.d.). Retrieved July 05, 2016, from [https://blogs.oracle.com/perrin/entry/the\\_lumberjack](https://blogs.oracle.com/perrin/entry/the_lumberjack)

---

# Module #3

## Armed with the theoretical knowledge acquired in the previous workshops, create a ZFS FreeBSD based File Server.

Now we know about RAIDZ, ZIL, cache, vdevs, etc. Now it is time to do something useful with this knowledge and start serving content using the mighty ZFS.

To start exposing our pools to external clients, we will use NFS for our Unix users and Samba for our Windows users.

### Installing Samba

Samba allows us to share files using the SMB/CIFS protocol used by Windows machines.

*“Since 1992, Samba has provided secure, stable and fast file and print services for all clients using the SMB/CIFS protocol, such as all versions of DOS and Windows, OS/2, Linux and many others.*

*Samba is an important component to seamlessly integrate Linux/Unix Servers and Desktops into Active Directory environments. It can function both as a domain controller or as a regular domain member.”*

We could install Samba from ports:

```
# cd /usr/ports/net/samba44/ && make install && make clean
```

or just use pkg install.

```
# pkg install samba44-4.4.5_1
```

After Samba has finished installing, enable it to be started at boot using SYSRC(8) or manually editing /etc/rc.conf

# Module #3

```
# sudo sysrc -f /etc/rc.conf samba_server_enable=YES
```

Create your smb4.conf file in:

```
/usr/local/etc/smb4.conf,
```

using the following configuration:

```
[global]

workgroup = HOMELAB

server string = SMB CIFS FREEBSD

security = user

load printers = no

guest account = nobody

log file = /var/log/samba/log.%m

log level = 2

max log size = 50

hide dot files = yes

socket options = TCP_NODELAY SO_RCVBUF=131072 SO_SNDBUF=65536

use sendfile = yes

strict locking = no

follow symlinks = yes

wide symlinks = yes

unix extensions = no

;; Our shares
```

# Module #3

```
[samba]

path=/zroot/shared

browseable = yes

writable = yes

valid users = nfs

create mask = 0660

directory mask = 0770
```

In my case, I will use a data set called shared; you must use your newly created pool.

More information on this config file is available in the following url:  
<https://www.samba.org/samba/docs/man/manpages/smb.conf.5.html>

Let's start the server.

```
# sudo service samba_server restart
```

Now we need to create a user to access this shared dataset, as we are going to also use NFS on this dataset. We will create a user called nfs then.

```
# pw useradd nfs
```

Change userid of user to 1000. Why? Because NFS user IDs are resolved on the client's side. Execute the following command and change the nfs user id to 1000:

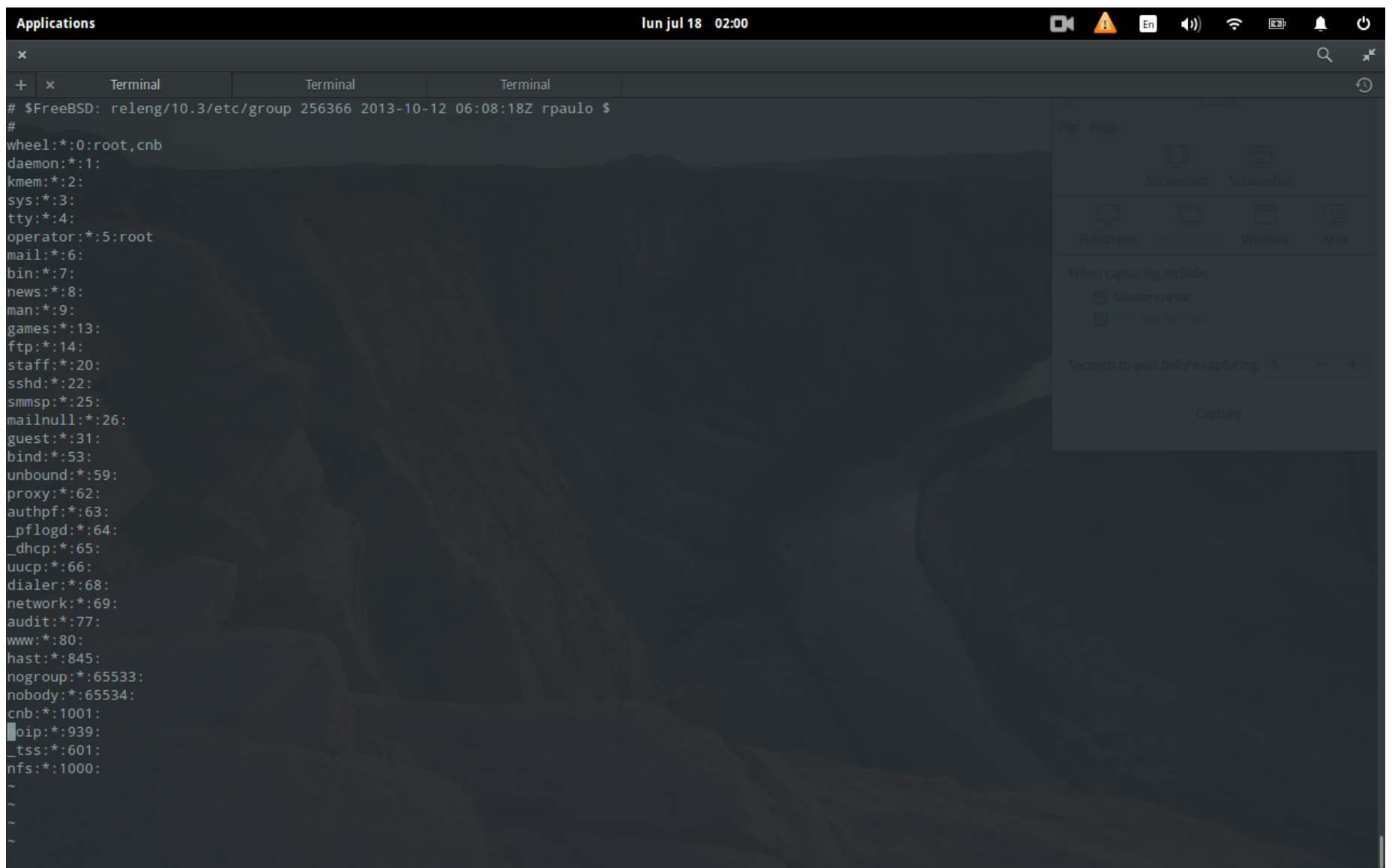
```
# vipw
```

# Module #3

A screenshot of a FreeBSD system's user list output in a terminal window. The window title is "Applications" and the system clock shows "lun jul 18 01:58". The terminal output lists system users and their properties, including UID, GID, home directory, and shell. The users listed are: root,toor,daemon,operator,bin, tty,kmem,games,news,man,sshd,smmsp,mailnull,bind,unbound,proxy,\_pflogd,\_dhcp,uucp,pop,auditdistd,www,hast,nobody,cnb,noip,\_tss, and nfs. The output is truncated with tildes (~) at the bottom. The terminal window has a dark theme and a search icon in the top right corner.

Then modify the nfs group to 1000:

# Module #3



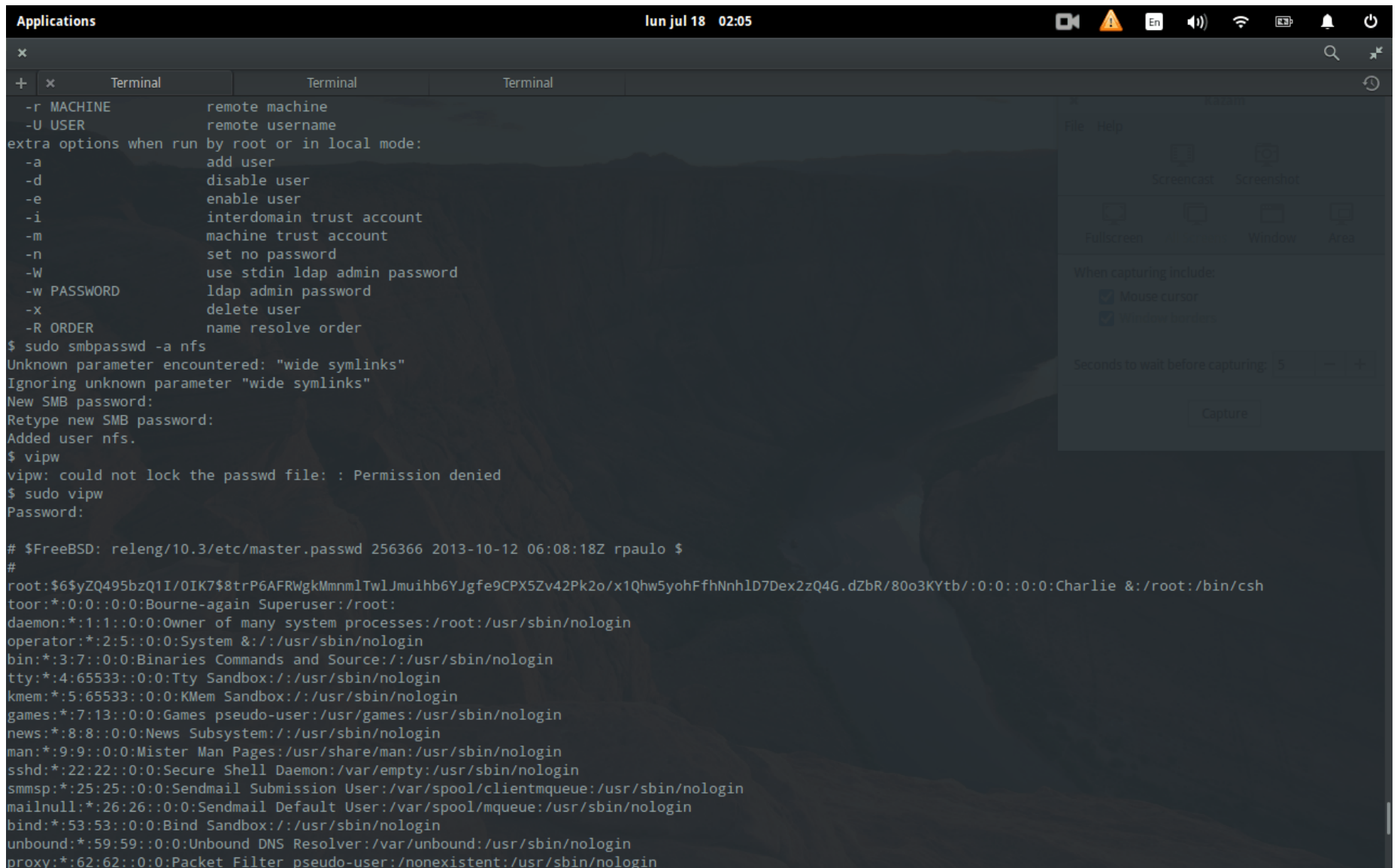
Now change ownership of the recently created dataset to `nfs:nfs`:

```
# chown nfs:nfs /zroot/shared
```

Finally, with the Samba server running, add this user to Samba so we could share the same dataset using Samba and nfs.

```
# smbpasswd -a nfs
```

# Module #3



```
Applications                               lun jul 18 02:05
+ x Terminal Terminal Terminal
-r MACHINE remote machine
-U USER remote username
extra options when run by root or in local mode:
-a add user
-d disable user
-e enable user
-i interdomain trust account
-m machine trust account
-n set no password
-W use stdin ldap admin password
-w PASSWORD ldap admin password
-x delete user
-R ORDER name resolve order
$ sudo smbpasswd -a nfs
Unknown parameter encountered: "wide symlinks"
Ignoring unknown parameter "wide symlinks"
New SMB password:
Retype new SMB password:
Added user nfs.
$ vipw
vipw: could not lock the passwd file: : Permission denied
$ sudo vipw
Password:

# $FreeBSD: releng/10.3/etc/master.passwd 256366 2013-10-12 06:08:18Z rpaulo $
#
root:$6$yZQ495bzQ1I/0IK7$8trP6AFRWgkMmmlTwlJmuhb6YJgfe9CPX5Zv42Pk2o/x1Qhw5yohFfhNnh1D7Dex2zQ4G.dZbR/80o3KYtb/:0:0:0:Charlie &:/root:/bin/csh
toor:*:0:0:0:Bourne-again Superuser:/root:
daemon:*:1:1:0:0:Owner of many system processes:/root:/usr/sbin/nologin
operator:*:2:5:0:0:System &:/usr/sbin/nologin
bin:*:3:7:0:0:Binaries Commands and Source:/usr/sbin/nologin
tty:*:4:65533:0:0:Tty Sandbox:/usr/sbin/nologin
kmem:*:5:65533:0:0:KMem Sandbox:/usr/sbin/nologin
games:*:7:13:0:0:Games pseudo-user:/usr/games:/usr/sbin/nologin
news:*:8:8:0:0:News Subsystem:/usr/sbin/nologin
man:*:9:9:0:0:Mister Man Pages:/usr/share/man:/usr/sbin/nologin
sshd:*:22:22:0:0:Secure Shell Daemon:/var/empty:/usr/sbin/nologin
smmsp:*:25:25:0:0:Sendmail Submission User:/var/spool/clientmqueue:/usr/sbin/nologin
mailnull:*:26:26:0:0:Sendmail Default User:/var/spool/mqueue:/usr/sbin/nologin
bind:*:53:53:0:0:Bind Sandbox:/usr/sbin/nologin
unbound:*:59:59:0:0:Unbound DNS Resolver:/var/unbound:/usr/sbin/nologin
proxy:*:62:62:0:0:Packet Filter pseudo-user:/nonexistent:/usr/sbin/nologin
```

Now install a Samba client, like `net/samba-smbclient` if you are on FreeBSD, or just try to browse to this shared folder using a Windows machine, you should be able to create files in that dataset.

## Sharing ZFS using NFS

Sharing via NFS is not problem at all, just use the `share nfs` keyword in your `pool/dataset`

```
# zfs share nfs="-network X.X.X.X -mask 255.255.255.0" pool/dataset/
pool/dataset -alldirs 192.168.1.x/24
```

This will make this dataset available via NFS to anyone with access to your remote machine, if you want to avoid that you could specify the network allowed to connect to your dataset.

```
# zfs share nfs="-network X.X.X.X -mask 255.255.255.0" pool/dataset
```

Modify or create `/etc/exports` with the following content (you need to change the `pool/dataset` and network to reflect your environment)

# Module #3

```
/pool/dataset -alldirs 192.168.1.x/24
```

Now you need to start your nfs daemon if it is not running for this to work, add the following option to `/etc/rc.conf` to start the nfs daemon at boot.

```
# sysrc -f /etc/rc.conf nfs_server_enable="YES"
```

To enable NFSv4 we need to add also:

```
# sysrc -f /etc/rc.conf nfsv4_server_enable="YES"
# sysrc -f /etc/rc.conf nfsuserd_enable="YES"
# sudo mount -o v3 <hostip>:/zroot/shared /mnt
```

Now connect to your dataset with the following command:

```
# sudo mount -o v3 <hostip>:/zroot/shared /mnt
```

You should have mounted now the zfs dataset in your client.

## Managing snapshots with zfstools

If you remember, on module 2 we classified ZFS properties in native and custom ones. And custom properties usually were created for administrative purposes or just internal conventions. Well `zfstools` <https://github.com/bdrewery/zfstools> uses custom set properties to manage if a snapshot is created or not.

Let's create some snapshots of our data using `zfstools`. First we need to install `zfstools`:

```
# pkg install zfstools
```

`zfstools` uses the ZFS property `com.sun:auto-snapshot` to determine if it should snapshot a filesystem. To see if the property is set on a dataset, we use the `zfs get` command.

```
# zfs get com.sun:auto-snapshot
# zfs set com.sun:auto-snapshot=true zroot/shared
```

# Module #3

This tool uses crontab to do its magic, so we will add the recommended crontab to our file server:

```
# crontab -e
```

paste this into crontab (type i then paste the content) then, to save and exit press escape then type :wq

```
15,30,45 * * * * root /usr/local/sbin/zfs-auto-snapshot frequent 4
0 * * * * root /usr/local/sbin/zfs-auto-snapshot hourly 24
7 0 * * * root /usr/local/sbin/zfs-auto-snapshot daily 7
14 0 * * 7 root /usr/local/sbin/zfs-auto-snapshot weekly 4
28 0 1 * * root /usr/local/sbin/zfs-auto-snapshot monthly 12
```

This is the recommended frequency of the auto snapshots, to execute one at demand, we just type the commands specified in the crontab. From the github site usage is the following (<https://github.com/bdrewery/zfstools>).

## Usage

```
/usr/local/sbin/zfs-auto-snapshot INTERVAL KEEP
```

- **INTERVAL** – The interval for the snapshot. This is something such as frequent, hourly, daily, weekly, monthly, etc.
- **KEEP** – How many to keep for this INTERVAL. Older ones will be destroyed.

Typing the following:

```
# zfs-auto-snapshot frequent 4
```

# Module #3

will create a snapshot named `zfs-auto-snap_frequent` concatenated with a timestamp

| NAME  | USED | AVAIL   | REFER | MOUNTPOINT |
|---|------|---------|-------|------------|
| <code>zroot/shared@zfs-auto-snap_frequent-2016-07-19-11h40</code> | 8K   | - 156K  | -     |            |
| <code>zroot/shared@zfs-auto-snap_frequent-2016-07-19-11h45</code> | 8K   | - 156K  | -     |            |
| <code>zroot/shared@zfs-auto-snap_frequent-2016-07-19-11h50</code> | 88K  | - 1.45M | -     |            |
| <code>zroot/shared@zfs-auto-snap_frequent-2016-07-19-12h12</code> | 8K   | - 15.1M | -     |            |

If you want to rollback to any version of the snapshots you have created, just use the `rollback` parameter.

```
# zfs rollback zroot/shared@zfs-auto-snap_frequent-2016-07-19-12h12
```

You could always run `snapshot`, but `zfstools` makes things easier to administrate.

```
# zfs snapshot zroot/shared@yourtag
```

## Check ZFS I/O

When we are serving content on our file server, we will sometimes want to check the IO statistics at a given time, for that we have `iostat` parameter to `zpool`:

|       |  | capacity |       | operations |       | bandwidth |       |
|-------|--|----------|-------|------------|-------|-----------|-------|
| pool  |  | alloc    | free  | read       | write | read      | write |
| ----- |  |          |       |            |       |           |       |
| rz1   |  | 496K     | 2.98G | 0          | 0     | 691       | 250   |
| zroot |  | 3.11G    | 4.83G | 0          | 1     | 13.1K     | 24.7K |
| ----- |  |          |       |            |       |           |       |

# Module #3

Where each column represents the following:

## alloc capacity

The amount of data currently stored in the pool or device. This amount differs from the amount of disk space available to actual file systems by a small margin due to internal implementation details.

For more information about the differences between pool space and dataset space.

## free capacity

The amount of disk space available in the pool or device. As with the used statistic, this amount differs from the amount of disk space available to datasets by a small margin.

## read operations

The number of read I/O operations sent to the pool or device, including metadata requests.

## write operations

The number of write I/O operations sent to the pool or device.

## read bandwidth

The bandwidth of all read operations (including metadata), expressed as units per second.

## write bandwidth

The bandwidth of all write operations, expressed as units per second.

If we want to check IO statistics in a given time frame for example, to check IO every 2 seconds.

```
# zpool iostat <mypool> 2
```

You could also specify a count parameter so the statistics will stop at the nth iteration.

```
# zpool iostat <mypool> 2 3
```

This will run statistics 2 times every 2 seconds.

Also we could check statistics by disk using `zpool iostat`.

```
# zpool iostat <mypool> -v
```

# Module #3

```
$ zpool iostat -v
```

| capacity           | operations | bandwidth |      |       |       |       |  |
|--------------------|------------|-----------|------|-------|-------|-------|--|
| pool               | alloc      | free      | read | write | read  | write |  |
| -----              | ---        | ---       | ---  | ---   | ---   | ---   |  |
| rz1                | 496K       | 2.98G     | 0    | 0     | 462   | 167   |  |
| raidz1             | 496K       | 2.98G     | 0    | 0     | 17    | 152   |  |
| /home/cnb/gooddisk | -          | -         | 0    | 0     | 439   | 88    |  |
| /home/cnb/diskz2   | -          | -         | 0    | 0     | 306   | 89    |  |
| /home/cnb/diskzp   | -          | -         | 0    | 0     | 308   | 89    |  |
| logs               | -          | -         | -    | -     | -     | -     |  |
| /home/cnb/logdisk  | 4K         | 1008M     | 0    | 0     | 444   | 14    |  |
| -----              | ---        | ---       | ---  | ---   | ---   | ---   |  |
| zroot              | 3.11G      | 4.83G     | 0    | 1     | 8.75K | 21.3K |  |
| ada0p3             | 3.11G      | 4.83G     | 0    | 1     | 8.75K | 21.3K |  |
| -----              | ---        | ---       | ---  | ---   | ---   | ---   |  |

## References:

28.10. File and Print Services for Microsoft® Windows® Clients (Samba). (n.d.). Retrieved July 18, 2016, from <https://www.freebsd.org/doc/handbook/network-samba.html>

28.3. Network File System (NFS). (n.d.). Retrieved July 18, 2016, from [https://www.freebsd.org/doc/en\\_US.ISO8859-1/books/handbook/network-nfs.html](https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/network-nfs.html)

(2015, February 28). Jailed ownCloud in FreeBSD 10.1. Retrieved July 17, 2016, from <https://l33t.codes/jailed-owncloud-in-freebsd-10-1/>

FreeBSD ZFS guide. (n.d.). Retrieved July 17, 2016, from <http://zfsguru.com/doc/bsd/zfs>

# Module #3

FreeBSD ZFS snapshots with zfstools. (2014, August 06). Retrieved July 17, 2016, from <http://blather.michaelwlucas.com/archives/2140>

NFS and ZFS, a fine combination (Bizarre ! Vous avez dit Bizarre ?). (n.d.). Retrieved July 24, 2016, from [https://blogs.oracle.com/roch/entry/nfs\\_and\\_zfs\\_a\\_fine](https://blogs.oracle.com/roch/entry/nfs_and_zfs_a_fine)

Opening Windows. (n.d.). Retrieved July 17, 2016, from <https://www.samba.org/>

Owncloud, the magic of ZFS snapshots, and the Recovery of Encrypted Files | Juan A. Sillero. (2015, October 21). Retrieved July 17, 2016, from <http://torroja.dmt.upm.es/jsillero/?p=726>

Smb.conf. (n.d.). Retrieved July 18, 2016, from

<https://www.samba.org/samba/docs/man/manpages/smb.conf.5.html>

---

# About the Instructor



Carlos Neira has worked about ten years as a software developer, porting and debugging enterprise legacy applications in several languages, like C, C++, Java, Common Lisp, Clojure and Python. He is currently employed as a software developer under Z/OS, debugging and troubleshooting legacy applications for a global financial company. In his free time, he tries to contribute to the PC-BSD project and enjoys metal detecting.

## About the Course:

**“Using FreeBSD as a File Server With ZFS”** you will learn how to use the current ZFS capabilities to help us build a home file server using FreeBSD 10.3.

Find it here <https://bsdmag.org/course/using-freebsd-as-a-file-server-with-zfs-2/>

## Teaser video:



<https://youtu.be/DxRpUW2z6vs>